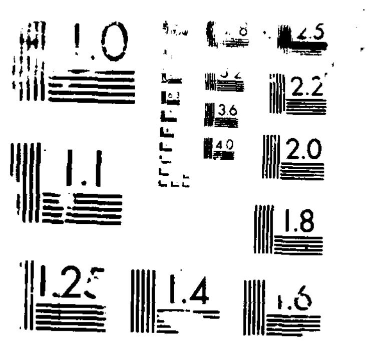
AD-A189 199 IPC FOR MODULAR SOFTWARE REQUIRES A THIRD PARTY CONNECT 1/1
(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE
S A FRIEDBERG JUN 87 TR-220 DACA76-85-C-0001

UNCLASSIFIED F/G 12/7 NL



[illegible]SOL[illegible]ION T[illegible]HART

IPC for Modular Software Requires a Third Party Connect

Stuart A. Friedberg
Computer Science Department
University of Rochester
Rochester, New York 14627

TR 220
June 1987

rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

(4)

IPC for Modular Software Requires a Third Party Connect

Stuart A. Friedberg
Computer Science Department
University of Rochester
Rochester, New York 14627

TR 220
June 1987

Abstract

Reconfiguration, on-line maintenance, load balancing, and similar operations on complex software require (among other things) dynamic binding of communicating peers, so that modules can be installed and removed in a framework of other functional modules. A binding may take the form of TCP/IP protocol connection, remote procedure call server location, shared memory address resolution, or object handle resolution, depending on the underlying communication mechanism. However, the need for abstraction and modularity is independent of the particular IPC mechanism(s) being used. A module should not need to know how it is used, specifically, how it is bound together with other modules to implement a more complex function.

Therefore, the bindings between two modules *A* and *B* should in general be created and destroyed by a third module *C*. We call this facility a *third party connect*. The module *C* should not have to have any particular relationship (other than authentication) to modules *A* and *B* in order to dynamically connect and disconnect them. In particular, it should not have to be one of *A* and *B*, be a common ancestor of them, or have an existing binding to them. Similarly, neither *A* nor *B* should have to take an active part in establishing or tearing down bindings.

Most IPC mechanisms either do not support third party connects at all (e.g., TCP/IP, Accent), do not allow dynamic reconfiguration (e.g., UNIX), or give binding authority to only a fixed internal element of the system (e.g., remote procedure call). Designers of future network communication protocols and interprocess communication mechanisms should provide for more flexible session layer features in general, third party connect in particular.

This work was supported in part by U.S. Army Engineering Topographic Laboratories research grant no. DACA76-85-C-001. We thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 220 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER A189 199
4. TITLE *(and Subtitle)* IPC for Modular Software Requires a Third Party Connect	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stuart A. Friedberg	8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE June 1987
	13. NUMBER OF PAGES 11
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, VA 22217	15. SECURITY CLASS. *(of this report)* Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Special
A-1	

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Binding, protocol connection, inter-process communication, third party connect , reconfiguration, modularity, survivable applications, distributed software

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Reconfiguration, on-line maintenance, load balancing, and similar operations on complex software require (among other things) dynamic binding of communicating peers, so that modules can be installed and removed in a framework of other functional modules. A binding may take the form of TCP/IP protocol connection, remote procedure call server location, shared memory address resolution, or object handle resolution, depending on the underlying communication mechanism. However, the need for abstraction and modularity is independent of the particular IPC mechanism(s) being used. A module should

20. ABSTRACT (Continued)

not need to know how it is used (specifically, how it is bound together with other modules to implement a more complex function).

Therefore, the bindings between two modules A and B should in general be created and destroyed by a third module C. We call this facility a third party connect. The module C should not have to have any particular relationship (other than authentication) to modules A and B in order to dynamically connect and disconnect them. In particular, it should not have to be one of A and B, be a common ancestor of them, or have an existing binding to them. Similarly, neither A nor B should have to take an active part in establishing or tearing down bindings.

Most IPC mechanisms either do not support third party connects at all (e.g., TCP/IP, Accent), do not allow dynamic reconfiguration (e.g., UNIX), or give binding authority to only a fixed internal element of the system (e.g., remote procedure call). Designers of future network communication protocols and interprocess communication mechanisms should provide for more flexible session layer features in general, third party connect in particular.

1. Introduction

Many investigations of distributed programming start with a particular interprocess communication mechanism and later explore its impact on structuring distributed programs. The University of Rochester Hierarchical Process Composition project (HPC) began with a particular structuring mechanism, and explored its impact on interprocess communication [LeF85b], [LeF85a]. Some of the results are quite general, and here we report an observation that applies to most software systems that emphasize abstraction or modularity and allow dynamic reconfiguration.

Most modern software systems build up complex applications from a collection of more-or-less independent modules implementing different functions. In various systems such modules might be Ada packages, Eden object, heavyweight processes, etc., and the intermodule (interprocess) communication mechanisms used to bind them together obviously vary as widely. Although our observations took place in the specific context of processes and transport protocols, we will use the generic terms module and IPC mechanism.

There are three distinct areas of responsibility in interprocess or intermodule communication: *signalling*, *composition*, and *implementation*. These functions take different forms depending on the communication mechanism being used, but in general, signalling defines the interactions between communicating peers (*what*), composition defines which peers communicate (*with whom*), and implementation provides the media of communication (*with what*).

We will not have much to say about signalling, except that it is a completely distinct function from composition. The interesting observations are, in a system supporting abstraction and reconfiguration:

(1) Composition is distinct from implementation.

(2) Composition is an incremental and distributed function.

(3) Implementation requires a third party connect.

2. Signalling, Composition, and Implementation are Distinct Functions

It is important to distinguish *signalling*, *composition*, and *implementation*. A signalling operation is an act of communication, like sending a message, writing to a file, or invoking an operation on an object. A signalling sequence defines the basic behavior of a process. Composition defines how basic behaviors are combined into a more complex application, by controlling the communication paths among its components. Implementation establishes the physical media needed to support the logical paths determined by composition.

While the correspondence is not exact, in terms of the ISO seven-layer model, signalling is user invocation of the transport layer, composition is user invocation of the session layer, and implementation is carried out by the session layer. Tables 1 and 2 illustrate the signalling and composition operations, respectively, for several interprocess communication mechanisms.[1] These Tables are meant to be suggestive, rather than

Mechanism	Signalling Operations
CONIC	send, receive, reply
Hydra	send, receive, reply
RPC	call, accept-reply
socket	*various, usually* send, receive
memory	read, write, P, V, fetch-and-phi
file	read, write, seek
mailbox	deposit, withdraw
link	send, receive
filter	send, receive
Linda	insert, remove, retrieve, eval-and-expand

Table 1.

Mechanism	Composition Operations
CONIC	link, unlink
Hydra	connect, disconnect
RPC	bind
socket	bind, listen, accept, connect, disconnect
memory	link, load, address
file	create, open, close, inherit
mailbox	create, name
link	create, transfer
filter	set-filter
Linda	set-pattern

Table 2.

exhaustive.

As shown by the Tables, different mechanisms generally have different signalling operations. For example, signalling with a message passing mechanism involves deciding

[1] Descriptions of these mechanisms may be found in: CONIC: [KMS83]; Hydra: [WLH81]; RPC: [Lis80], [Nel81]; socket: [CCC70]; mailbox: [Bri73], [BFL76], [HLG78], [JCD79]; link: [BHM77], [RaR81], [ZwL83], [ACF87] (Accent terminology for link is "port", not to be confused with the Hydra and CONIC ports); broadcast filters: [FFH73], [Ary81], [KeS84], [GKZ85]; Linda: [Gel85].

when to send and receive messages, and what the contents of messages should be. Signalling with a semaphore mechanism involves deciding when to wait (*P*) and signal (*V*). Signalling with a remote procedure call mechanism involves deciding when to call a procedure, when to return from a call, and what the arguments and return values should be.

The compositional operations for these three example mechanisms are deciding whom to send a message to (whom to receive from), deciding which processes have access to the semaphore, and deciding which client stubs are bound to which server entries, respectively. The implementation of conventional interprocess communication mechanisms is usually triggered directly by composition and managed by the host operating system.

Dividing an application into many modules reduces complexity by providing many small, specific interfaces with a small number of possible clients. Interfaces are usually talked about as static specifications, but it is a module's behavior or pattern of interactions that is important, not its entry points and arguments. Each module expresses a behavior through its signalling with related modules, and in systems that enforce abstraction, this behavior will not depend on the identity of a communicating peer module. That is, signalling should not depend on composition.

Dynamic installation, replacement, or removal of a module from a running system requires (among other things) the ability to change the bindings between communicating modules. Upgrading a system on-line requires, for example, the ability to remove all communication paths to an outdated module, remove it, create a replacement, and then create communication paths from the old modules peers to the new module. To provide general purpose tools for reconfiguring applications on the fly, composition should not require the active cooperation of the affected modules. That is, dynamic composition should not depend on signalling, or on the identities of the composed modules.

We are concerned mainly with how processes can be dynamically reconfigured to support flexible, long-lived, survivable applications. This means we are not interested in signalling *per se*, but very interested in composition as a logical operation and implementation as a physical operation. The HPC project has a specific notion of how to form logical communication paths between processes and groups of processes, but our observations will be phrased in terms of more general systems.

3. Composition is Incremental and Distributed

Signalling, composition and implementation are not just conceptually different activities. If distributed programming is to incorporate more complex abstractions than the trivial client-server model, these activities must actually be carried out by different agents. We argue that composition *must* be an activity distributed among multiple agents. Further, the composing agents are generally *not* the same as the signalling agents.

We are interested in software systems with three significant properties. An interesting system (1) allows communication with a group of related modules (processes) as a group, (2) enforces the abstraction of a group, and (3) supports non-trivial internal structures in a group.[2] In such a system,

(1) logical communication paths are naturally divided into several segments, one for each group that the path involves, and

(2) composition is a distributed activity.

Readers who object to "path" as too suggestive of physical routing are encouraged to substitute "binding decision". The difference between binding decision and binding is just that between composition and implementation.

There is no shortage of interesting systems. The following list of examples could be doubled or tripled in length easily. Allowing for vagaries of notation, Figure 1 is a natural illustration for the kind of nested structure that can be described, and in some cases implemented directly, by each of the example systems.

- distributed systems –
  CONIC [KMS83], HPC [LeF85a]
- programming languages and environments –
  DPL-82 [Eri82], Pict [GIT84], PRONET [LeM82]
- system design, analysis, and modelling tools –
  DREAM [Rid81], GRACE/CS [HaK84], SADT [Ros85], SARA [FFR86], SREM [Alf85]

The direct paths between modules at the same level of abstraction, combined by the paths inside modules that connect implementing children to the external interfaces of the

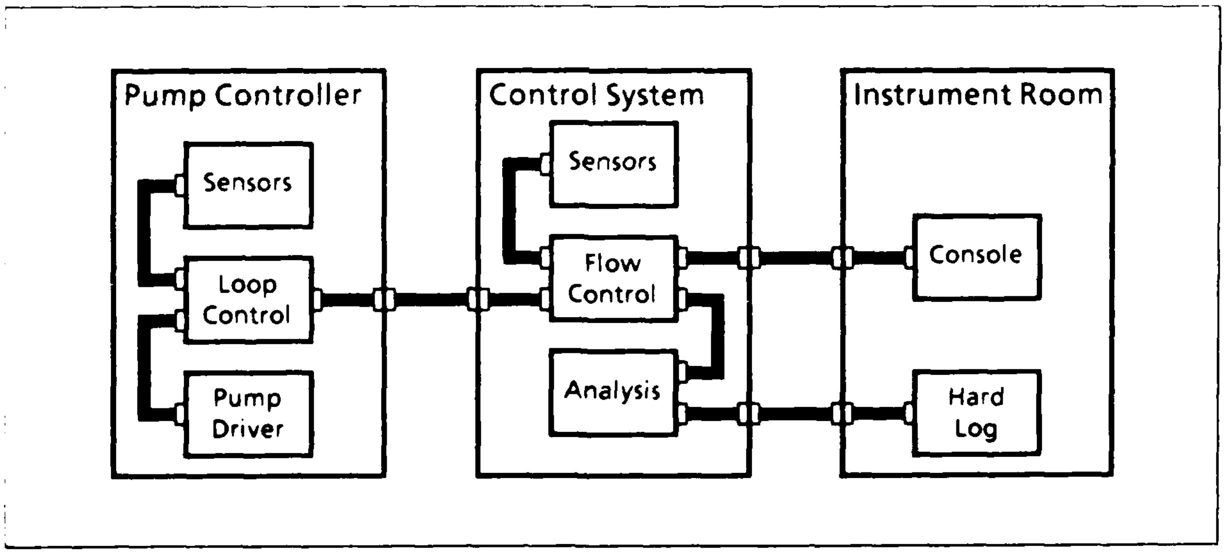


Figure 1.

[2] Either nesting of groups or designating specific modules to receive communication addressed to a group satisfies property 3. Systems with only single level of process grouping and uniform internal treatment of processes within a group (e.g., multicasting), such as V [ChZ85], lack property 3.

module, naturally produce incremental compositions. In most cases, different agents will be responsible for the design or run-time management of different abstractions, leading to distribution of composition, as well.

Even simpler and more common systems demonstrate incremental composition. Consider a system offering only one service with multiple server processes, a file service, say. Every process either belongs to, or is a client of, the file service, and the multiprocess implementation of the file service is transparent to the clients. Before a client and a server process communicate, the client must decide to access the file service and some agent in the file service must decide which server process is to handle the client's request. Neither the client, nor the file service agent, can decide unilaterally to bind the specific client and server processes. The logical path between the communicating processes has two segments. (There are two independent contributions to the decision to bind that particular pair of processes).

Not all our example systems support reconfiguration, and static structure avoids run-time composition altogether. Composition is then typically an activity of human designers, while implementation is carried out by configuration software. Signalling remains strictly a run-time activity. However, static structure makes the question of distributed and incremental composition a moot issue of design methodology. We suggest that in successful methodologies composition remains distributed and incremental when the metric involves separation among specification modules rather than process groups or nested modules.

4. Efficient Implementation Requires a Third Party Connect

Robust, flexible, distributed software will be, and is, designed as applications with several levels of internal abstraction and significant, long-lived, internal communication patterns. To allow designer and maintainers run-time access to the relationships among an application's components, the logical grouping and communication paths should be explicit and persistent. However, while this useful structure should be kept at run-time, it is not necessary or desirable to use all of it when manipulating physical resources like processes and communication channels.

CONIC, HPC, PRONET, and the other examples operating at run-time provide a value-added service by supporting intermediate levels of abstraction and incremental definition of communication paths. If all the purely abstract structure is eliminated by erasing the abstract grouping and throwing away all the incomplete paths, the remaining structure consists of real modules (processes) and real communication media. Such an elimination is illustrated in Figure 2. Ignoring abstract structure in this way has *no* effect on the behavior of the system. No real modules, or paths between real modules, have been removed.

So, on the one hand, there is the user model with lots of convenient abstraction, and on the other, there is the physical environment with only real pieces. Some agency, for convenience we will call it an operating system,[3] translates the incrementally composed logical paths of the user model into the appropriate physical communication channels.

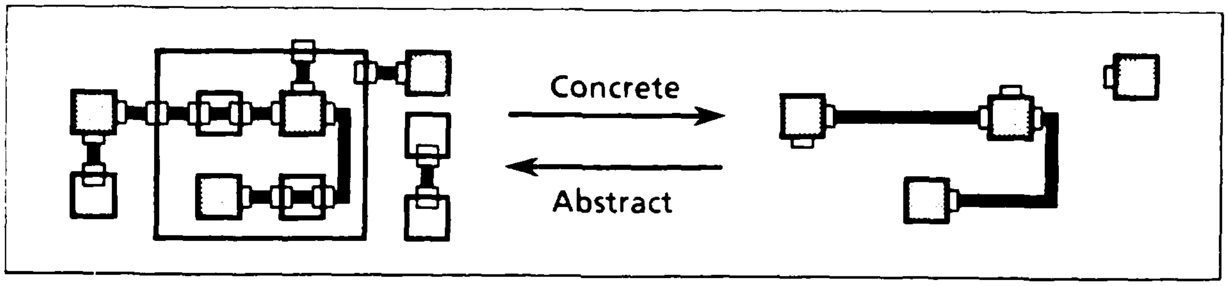


Figure 2.

This is the third area of responsibility: implementation.

Clearly, the only acceptable implementations are those that create physical media directly between the real communicating modules, without additional overhead, and regardless of the length and complexity of the logical path joining them. This is practically a definition of a third-party connect. After the operating system has stripped away the user level abstractions, it must be able to connect and disconnect directly any of the modules under its aegis. This facility is critical for efficient implementation of all multiprocess software that emphasizes abstraction: *Ability to bind (and unbind) modules with a communication path may rest with an agency that never was, is, or will be bound to those modules or associated with that path.* In other words, it must be possible to connect and disconnect two processes from a third process.

5. Discussion

5.1. State of the Art

It is imperative to abandon IPC mechanisms where the decision to connect processes must be made by the processes being connected. It is unreasonable to develop complex multiprocess software with such mechanisms, because they make it impossible to introduce the appropriate use of abstraction. This is, sadly, the state of most internetwork transport protocols, like TCP/IP, where communicating processes have to explicitly cooperate in establishing connections. (To be fair, XNS Bulk Data Transfer is an internetworking protocol designed from the beginning to support third-party connect [Xer84].)

Another inadequate tool is inheritance of communication paths from a common ancestor. This is the mechanism used by the UNIX shells to implement pipelines, and in principle, more complex patterns of communication between processes. It does support modularity and abstraction, but unfortunately does not allow for later reconfiguration of an application, even to replace a failed process.

[3] This agency may provided in any number of different ways, depending on the extent to which applications can be dynamically reconfigured. In the examples we have cited, the agency is variously the operating system, a privileged server process, a run-time library, a system generation utility, and a compiler.

The third common technique is most clearly illustrated by link systems, where to create a path between processes *A* and *B*, process *C* must already have a path to each of them. Links support modularity and reconfiguration, but require the active cooperation of the processes being reconfigured. In particular, process *C* can volunteer a new path, but can not cause processes *A* and *B* to stop using an old path. (A link system allowing the owner of a link to transfer receive rights and to delete a link even if rights are held by other processes would not have this deficiency.) Link systems usually also use inheritance to get over the bootstrapping problem of providing a process its initial paths.

None of these techniques, self-composition, inheritance, or links, is an adequate tool for the efficient, dynamic reconfiguration of modular software, yet there is nothing fundamentally difficult about providing third-party connect, or similar functions. For example, the portion of the TCP/IP protocol that is concerned with setting up and tearing down connections (session layer issues) is clearly defined. It would be straightforward to modify TCP to negotiate the connect and disconnect subprotocols with (generally) a third party. The server "passive mode" should be separated from the socket/address multiplexing facility.

It is no coincidence that modern system design and analysis tools emphasize abstraction and modularity. Interesting systems in the sense of Section 3 will become only more common. Clearly, design methodology has outstripped the abilities of IPC mechanisms to support it efficiently. The reason is perhaps historical: software designers are painfully aware of the benefits of modularity and the pitfalls of the expedient solution. IPC designers are often trained in a hardware culture, specifically the long-haul circuit switching tradition, with a concern for performance and traditional services. However, we can do a great deal more with IPC than emulate telephones and terminals! Designers of new network protocols and IPC mechanisms will be doing application designers a disservice if they do not include session issues in general, and third-party connect in particular, in their considerations.

5.2. Authorization

Given the potential for unilateral, uncooperative change (of communication paths), the question of authorization and permission naturally arises. To some extent, the three techniques just discussed are limited precisely because they want to limit changes in composition to implicitly authorized processes. The HPC project provides a specific answer to the problem of authentication and authorization, but we do not attempt a general answer!

Instead, we will note that there are at least two places where the permissions for created or destroying a connection between processes can be checked. First, as applications incrementally build up logical paths, their permission to modify the corresponding pieces of (abstract) structure can be checked. Second, as the implementing agency attempts to create or destroy a real communication channel, its permission to modify that channel can be checked.

The designer of internetwork transport protocols will be primarily interested in the latter case. Consider a modified TCP/IP. When a would-be implementing agent attempts to negotiate a change to a connection, the protocol software can authenticate the agent. While the IP and ISO internets lack practical authentication services, the Xerox corporate internet has provided them for several years. In this way, the transport protocol providing the third-party connect needs to know nothing about the structure of the applications in which it is being used, and the worker processes in an application simply have to agree on a common implementing agent and so inform the protocol.

5.3. Why Persistent Logical Connections (Compositions)?

The primary objection to communication based on explicit, persistent connections is traditional, and based on the cost of establishing and destroying switched circuit virtual circuits in the telegraph and telephone industries. To a large extent, this limiting technology has been left behind, even by those industries, but stronger responses to the claim of excessive cost can be made when composition is separated from implementation.

First, changes in logical composition only have an effect on media when a complete logical path is created or destroyed. Therefore, many, if not most, reconfigurations will affect only abstract structure and incur no cost in set-up or tear-down of physical media.

Second, many IPC mechanisms are fundamentally connectionless. Consider datagrams where "connecting" simply means determining the destination address of an outbound datagram. Establishing a communication channel incurs no cost until a datagram is actually sent, and then no greater cost than sending any other datagram.

Thus, by separating composition from implementation, a software designer and maintainer can have the abstraction of explicit, persistent connections with the performance and cost of the most appropriate communication mechanism(s).

6. References

[Alf85] M. W. Alford, "SREM at the Age of Eight: The Distributed Computing Design System", *Computer 18*, 4 (April 1985), 36-46.

[ACF87] Y. Artsy, H. Chang and R. Finkel, "Interprocess Communication in Charlotte", *IEEE Software 4*, 1 (January 1987), 22-28.

[Ary81] A. K. Arya, "Super: Encapsulated Autonomous Distributed Computations on an Abstract Architecture", Ph.D. Thesis, University of Rochester, July 1981.

[BFL76] J. E. Ball, J. A. Feldman, J. R. Low, R. Rashid and P. Rovner, "RIG, Rochester's Intelligent Gateway: System Overview", *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 321-328.

[BHM77] F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in Demos", *Proceedings 6th Symposium on Operating Systems Principles*, West Lafayette, Indiana, 16-18 November 1977, 23-31.

[Bri73] P. Brinch-Hansen, *Operating Systems Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[CCC70] C. S. Carr, S. D. Crocker and V. G. Cerf, "Host-Host Communication Protocol in the ARPA Network", *Proceedings AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey, 5-7 May 1970, 589-597.

[ChZ85] D. R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel", Technical Report 85-13, Department of Computer Science, Rice University, February 1985.

[Eri82] L. W. Ericson, "DPL-82: A Language for Distributed Processing", *Proceedings 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, Florida, 18-22 October 1982, 526-531.

[EFR86] G. Estrin, R. S. Fenchel, R. R. Razouk and M. K. Vernon, "SARA (System ARchitects Apprentice): Modelling, Analysis and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering SE-12*, 2 (February 1986), 293-311.

[FFH73] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis and L. A. Rowe, "The Distributed Computing System", *Proceedings 7th Annual IEEE Computer Society International Conference*, February 1973, 31-34.

[Gel85] D. Gelernter, "Generative Communication in Linda", *Transactions on Programming Languages and Systems 7*, 1 (January 1985), 80-112.

[GlT84] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment", *Computer 17*, 11 (November 1984), 7-25.

[GKZ85] R. Gueth, J. Kriz and S. Zueger, "Broadcasting Source-Addressed Messages", *Proceedings 5th International Conference on Distributed Computing Systems*, Denver, Colorado, 13-17 May 1985, 108-115.

[HaK84] M. Harada and T. L. Kunii, "A Recursive Graph Theory as a Formal Basis for a Visual Design Language", *Proceedings IEEE Computer Society Workshop on Visual Language*, Hiroshima, 6-8 December 1984, 124-135.

[HLG78] R. C. Holt, E. D. Lazowska, G. S. Graham and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading, Massachussetts, 1978.

[JCD79] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl, "StarOS, A Multiprocessor Operating System for the Support of Task Forces", *Proceedings 7th Symposium on Operating Systems Principles*, Pacific Grove, California, Dec 1979, 117-127.

[KeS84] J. Kepecs and M. Solomon, "SODA: A Simplified Operating System for Distributed Applications", TR 527, University of Wisconsin - Madison, January 1984.

[KMS83] J. Kramer, J. Magee, M. Sloman and A. Lister, "CONIC: An Integrated Approach to Distributed Computer Control Systems", *IEE Proceedings 130-E*, 1 (January 1983), 1-10.

[LeM82] R. J. LeBlanc and A. B. Maccabe, "The Design of a Programming Language Based on Connectivity Networks", *Proceedings 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, Florida, 18-22 October 1982, 532-541.

[LeF85a] T. J. LeBlanc and S. A. Friedberg, "HPC: A Model of Structure and Change in Distributed Systems", *IEEE Transactions on Computers C-34*, 12 (December 1985), 1114-1129.

[LeF85b] T. J. LeBlanc and S. A. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems", *Proceedings 5th International Conference on Distributed Computing Systems*, Denver, Colorado, 13-17 May 1985, 26-34.

[Lis80] B. Liskov, "Remote Procedure Call", Distributed Systems Group Note 64, MIT Laboratory for Computer Science, June 1980.

[Nel81] B. Nelson, "Remote Procedure Call", CMU, Pittsburgh, PA-CS-81-119, May 1981. Ph.D. Thesis.

[RaR81] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel", *Proceedings 8th Symposium on Operating Systems Principles*, Pacific Grove, California, 14-16 December 1981, 64-75.

[Rid81] W. E. Riddle, "An Assessment of DREAM", in *Software Engineering Environments*, H. Hunke (editor), North-Holland, 1981, 191-221.

[Ros85] D. T. Ross, "Applications and Extensions of SADT", *Computer 18*, 4 (April 1985), 25-34.

[WLH81] W. A. Wulf, R. Levin and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

[Xer84] Xerox Corporation, "Appendix F Bulk Data Transfer", Addendum 1a to Xerox System Integration Standard 038112, Stamford, Connecticut, April 1984.

[ZwL83] W. Zwaenepoel and K. A. Lantz, "Perseus: Retrospective on a Portable Operating System", TR STAN-CS-83-945, Stanford University, February 1983.